SOFTWARE REUSE: ISSUES AND RESEARCH DIRECTIONS

by

Yongbeom Kim
Leonard N. Stern School of Business
New York University
New York, NY 10006

and

Edward A. Stohr
Leonard N. Stern School of Business
New York University
New York, NY 10006

June 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-15

ABSTRACT

Software reuse has been considered as a means to help solve the software development crisis. This paper surveys recent work based on the broad framework of software reusability research, and suggests directions for future research. We address general, technical, and non-technical issues of software reuse, and conclude that reuse needs to be viewed in the context of a total systems approach. We also envision a software system or reuse support system(RSS) that helps document and elucidate existing application systems so that the ideas and design decisions involved in their creation can be reused either in the context of maintenance or when building new systems.

1. INTRODUCTION

Organizations face many problems in software development including increased costs, delayed schedules, unsatisfied requirements, and software professional shortages. This situation is often referred to as the software development crisis. Increases in software development productivity and improvement in software quality are necessary to allow organizations to maximize the return on investment in information technology. The new business environment, which is characterized by increased competition, global markets, and the need to cut costs, makes this improvement in software development productivity even more important.

In this paper, we examine software reusability as a means to improve the process of software development and also the quality of the software produced. Software reuse refers to the use of previously developed software components in new applications. Traditionally, this has involved code reuse by other programmers in the same organization. A more general concept is to reuse the concepts or ideas underlying the software system as well as the code itself. These concepts and ideas include the outputs of earlier phases in the system life cycle such as knowledge about the purpose of the software, the business process that

it is to support, the functions that are to be provided, and so on. Some of these reusable objects may be reusable in a number of different application domains. An example is provided by the development of the open systems concept for standardizing user interfaces. Alternatively, it may be possible to capture the common knowledge underlying a whole domain. (e.g. a branch of law or medicine) so that this can be reused in a range of software applications. Finally, it is important to consider the breadth of use of the reusable components - whether software resources in this general sense are reused by the original developer on different projects, a group of developers within a single organization, or, as envisaged by the Department of Defense [DOD 86], by a large number of different organizations.

We refer to the most general concepts of reusability as outlined in the previous paragraph as "global reusability". Here, we concentrate on an intermediate concept of reusability, "widespread reusability" which we define as (1)reuse by other software developers within the same organization as well as the original developer, (2) reuse of objects produced by the systems analysis and design phases as well as code, (3) reuse of general and specific purpose software resources across a variety of application domains, and (4) reuse of software resources along a continuum of task types from maintaining existing systems to developing new software systems. The problems of standardization across organizations and the capture of the knowledge underlying a given domain are not addressed directly in this paper.

The search for effective methods of promoting software reuse has an economic basis. When software systems are developed with the concept of software reuse, fewer total lines of code may need to be written and also the amount of documentation and testing may be reduced. That is, software reuse should increase productivity. Increased productivity will reduce development cost and schedule overruns. Since reusable software resources have usually been rigorously tested and verified, software quality should also be improved by software reuse.

There are several survey papers concerning software reusability including [Sundfor 83], [Horowitz and Munson 84], [Seppanen 87], and [Biggerstaff and Richter 89]. The goal of this paper is to survey more recent work based on a broad framework of software reusability research as well as to provide directions for future research in software reusability.

2. A FRAMEWORK FOR SOFTWARE REUSE

There are many approaches to the concept of software reuse. To organize and place various concepts and models of reuse (or reusability research), a number of conceptual frameworks for software reuse have been proposed.

A framework which classifies the available technologies for reusability into two major groups, composition technologies and generation technologies, is proposed by [Biggerstaff and Richter 89]. We will discuss these technologies in more detail in Section 6. Another framework based on three research and development questions, what is being reused?, how should it be reused?, and what is needed to enable successful reuse?, is developed by [Freeman 87]. In Freeman's framework, five levels of reusable information (code fragments, logical structure, functional architecture, external knowledge (such as application domain knowledge and software development knowledge), and environmental knowledge related to organizational and psychological issues) are defined. For each of the five information levels, typical projects of three different expected payoff periods are identified to answer research and development questions. Other frameworks by [Horowitz and Munson 84] and [Jones 84] are based on the forms of reuse such as data, code, and design.

In this paper, the framework for software reusability research shown in Figure 1 is used to organize our discussion. This framework is inclusive in the sense that most issues in other frameworks are discussed.

In Figure 1, research on software reuse is divided into three groups according to the point of view: general issues, technical issues, and non-technical issues. General issues are classified into definitions and scope of software reuse and economic issues. Technical issues are classified into reuse methodologies and software approaches. Non-technical issues are classified into organizational issues and psychological issues.

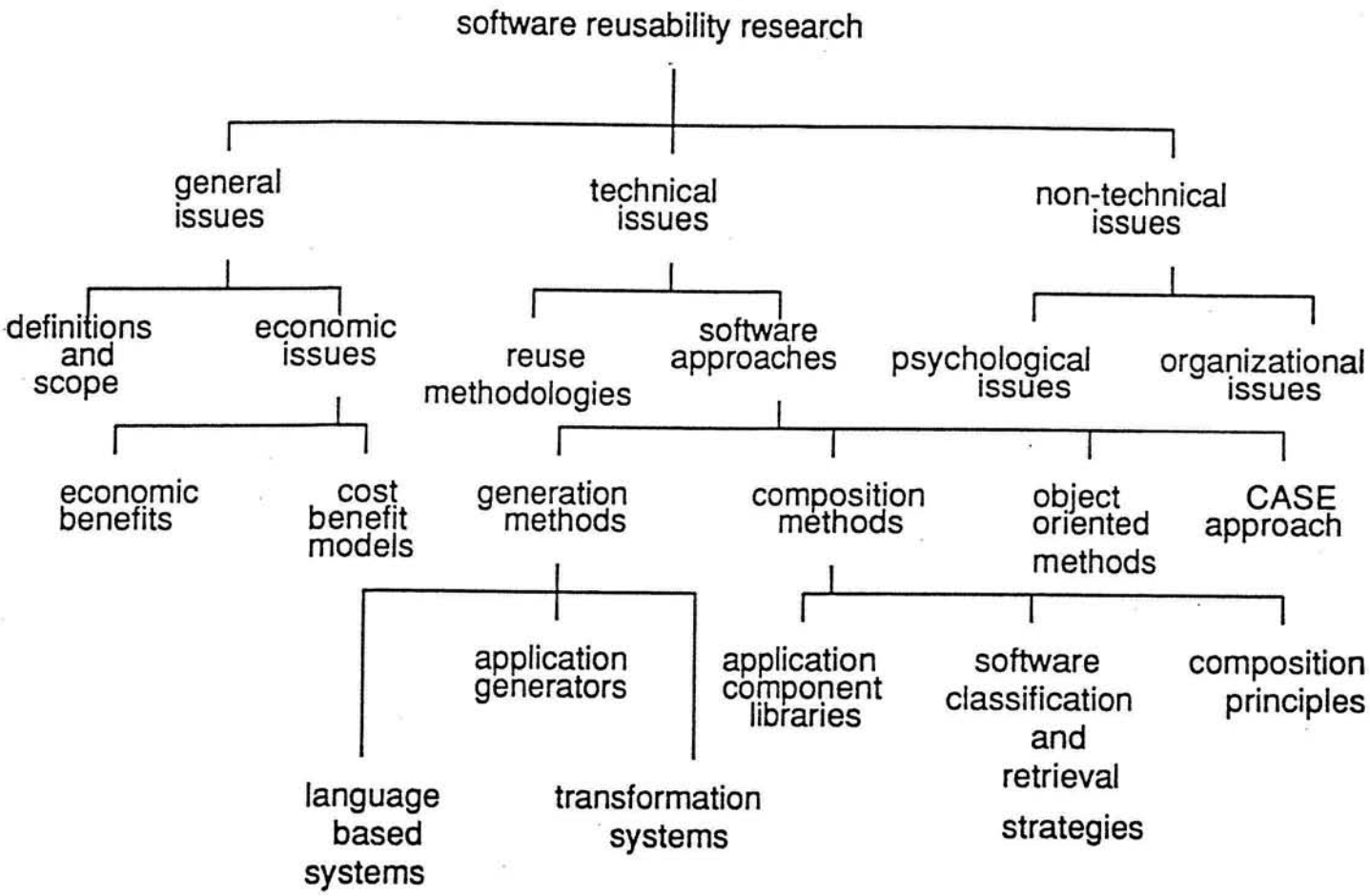

Figure 1. A Framework of Software Reusability Research

3. DEFINITIONS AND SCOPE OF SOFTWARE REUSE

Questions related to software reuse (such as what is software

reuse?, what do we reuse?, when do we apply software reuse?, and who reuses software?) have been considered by a number of researchers [Horowitz and Munson 84][Jones 84][Freeman 87][Tracz 90][Rubin 90]. Software reuse is defined as the use of previously developed software artifacts such as design, code, documentation, etc., in new applications by various users such as programmers and systems analysts.

To provide an organized and inclusive point of view, we define the concept of widespread software reuse with respect to the following criteria: user types, reusable resource types, and task types.

User Types

Users of reusable software resources can be classified into three groups: (1) the original developers, (2) individuals in the same organization, and (3) people in different organizations

When reusable software resources are well classified and easily retrievable, anyone in the same organization should be able to use them for software systems development. For organizations such as the Department of Defense that normally involve a number of different software contractors, reuse <u>across</u> different organizations can be extremely important both economically and from the point of view of developing sound, coherent, and maintainable systems [Myers 87] [DOD 86]. Software reuse by people in different organizations implies such problems as standardization and legal rights that will not be considered in this paper.

Reusable Resource Types

Reusable software resources can be classified by entity types, application area types, and abstraction level types as follows.

Entity Types

There are three kinds of entities that can be reusable: process, data, and object. An object resource is a combination of data and process resources [Wegner 90] [Micallef 88]. Note that in this paper, we focus on process and data resources only.

Process resources are usually considered the main target of software reuse. However, the importance of data resources as reusable software objects is recognized by the emergence of data base management systems(DBMS), standard data interchange format [Fylstra and Gill 80], expanded data definition which includes various data types such as graphics and voice, and many data-related applications.

Application Area Types

A given software resource (process or data) can exist in a wide range of contexts varying in a continuum from customized resources, to functional resources to generic resources.

A customized process resource is a set of application functions developed to satisfy the specific requirements of users in an organization. All of the software resources can be thought as working together to satisfy a set of organizational needs. Common examples are data processing systems for payroll, accounts receivable, etc. Examples of customized data resources are the files and screens used by customized software.

A functional process resource is a set of application functions that are packaged as a unit for a given application area such as management science, finance, accounting, or statistics. Each function of a functional collection can be used separately. Software packages such as IMSL [IMSL 84] and SPSSX [SPSS 86] fit in this category. Functional data resources are data definitions or data item values that are useful in an area of application. An example might be the data definitions in a data dictionary for a commonly used database of financial information.

A generic process resource is a general-purpose software resource that can be used in many different applications. Examples include file management, screen management, graphics, string management, print routines, keyboard management routines, help functions, editing routines, data entry routines, and date manipulation routines. Generic data resources include definitions and formats for items such as dates and personnel and product identifiers that are used across many applications. Generic or functional resources are used in customized resources.

Abstraction Levels

There are number of levels of abstraction, from abstract to concrete, at which both data and process entities may be considered. Representations of aggregations of processes into higher level subsystems or systems are at the abstract level. A process resource is at the concrete end of the spectrum, if it is in a form that can be directly used in a functioning software system, e.g., object code. In the middle, one has process entities on data flow diagrams, process narratives, or source code. Similarly, the abstract to concrete spectrum for data ranges from data objects in the conceptual models such as Entity-Relationship model to descriptions in data dictionaries to physical files through data declarations in programming language format.

It is generally agreed that the cost for coding is only a small portion of total software development cost [Freeman 87]. By reusing both concrete resources such as code, and abstract resources such as entity-relationship diagrams and data flow diagrams, software development costs should be reduced. Both modification of existing software resources and adaptation of existing software resources to new applications require understanding of existing reusable software resources. The existence of reusable resources at higher levels of abstraction can help the understanding process.

Task Types

The tasks related to software reuse can be classified along a continuum from maintaining existing systems to developing new software systems.

Maintenance includes two task types: modifying existing software systems and adding new components to enhance existing software systems. In both cases, maintenance can be viewed as a reuse-oriented task in which the appropriate requirements, design, and code from earlier versions of the system has to be accessed and understood by the maintenance programmer.

Three different task types are involved in developing new software systems. The classification is based on the degree of similarity of the application addressed by the new and old systems.

The first task type is building a new system that has process and data in common with an existing software system. An extreme example of this task type is building a new software system with the same functions as the existing software system but in a different implementation language. Both abstract level process and data resources from the old system are reused in this case. A less extreme example occurs when a completely new system is to be built that will replicate the logic of a part of the old system.

The second task type is building a new software system that performs an entirely different function but is related to an existing system because of common data. Here, the new system can reuse common files, file definitions, and screens. Thus, the reusable items are data entities, although "generic" software elements such as screen management routines that have been developed in the context of the first application might be reused in the second.

The third task type is building a completely independent new

software system. Here, the opportunities for reuse are probably limited to generic resources such as report writers, screen managers, and date routines.

Table 1 shows the potential of reuse of the various class of reusable resources over the different task types.

Entity Type	Data		Process	
Application Type	Generic ⟶	Customized	Generic ⟶	Customized
Task Type				
Maintenance	Low	High	Low	High
Developing new systems	↑	↑	↑	↑
common data and process related	↑	↑	↑	↑
common data related	↑	↑	↑	↑
completely independent	Low - Medium	Low	High	Low

Table 1. Likelihood of Reuse of Various Software Resources

By definition customized resources are likely to be "reused" in the maintenance task. Note that the reuse of previously developed ideas is almost total in a maintenance task. Because understanding of the existing system is essential, abstract resources from early phases in the life cycle can be important. On the other hand, for new application development, generic resources in the concrete level of abstraction (e.g., procedures for report writing, date routines, etc.) are more

likely to be important.

In summary, the concept of widespread software reuse is defined with equal emphasis on data and process resources, abstract and concrete resources, specific application-oriented and generic function-oriented resources. It also emphasizes a wide range of tasks from maintenance of existing systems to development of new systems.

4. ECONOMIC ISSUES

Reusing software resources can result in increases in productivity and improvements in quality [Standish 84] [Horowitz and Munson 84] [Boldyreff 89] [Rubin 90] and reliability [Lubars 86]. Economic issues concern (1) actual evidence of productivity and quality increase from software reuse and (2) cost benefit models of software reuse.

4.1 Economic Benefits from Software Reuse

Improving productivity is a major goal of software reuse, and is a key focus of many corporate IS groups [Frank 81]. Table 2 summarizes reported statistics about reuse rates and productivity increases from software reuse in organizations. Reuse rate is defined as the proportion of reused code in new systems. Productivity increase can be defined in terms of the number of lines of source code produced by programmers/unit-time, savings in man-months, or dollar savings.

Reference	Reuse Rate (%)	Projects	Productivity (Increase)
[Matsumoto 87]	48 %	multiple	8-9 % yearly increase
[Love 88]	25 %	single	save 130 man-months
[Conte 88]	60 %	multiple	50 % increase
[Joyce 88]	35 %	multiple	save 250 man-days/month
[Coome et al. 90]	42 %	multiple	33.9 non-comment lines/day
[Todd 90]	14 %	multiple	a savings of $1.5 million
[Banker and Kauffman 90]	65 %	multiple	see text
Average	41.3 %		

Table 2. Reported Statistics of Software Reuse

Unfortunately, there has been very little consistency or standardization in the productivity measures as can be seen in Table 2. It is therefore necessary to explain each case in some detail. Matsumoto reported 48 % code reuse rate in the Fuchu Factory of Toshiba Corporation in 1985; the average rate of yearly improvement of productivity was approximately 8-9 % between 1977 and 1985 [Matsumoto 87]. [Love 88] reported that 130 man-months was saved (from 460 man-months to 330 man-months) by reusing 50 components rather than building them from scratch for a 200 components development project using PPI's (Productivity Products International, Inc.) workplace tools for the Objective-C language. Raytheon is reported to have achieved 60 % reusable code in new development of COBOL applications and a 50 % increase in productivity at their Missile Systems Division in Bedford, MA [Conte 88]. The Hartford Insurance Group has a reusable library of 35 documented and tested COBOL code modules maintained on a Wang minicomputer [Joyce 88]; 35 % of the code in new systems comes from the reusable library. By reusing code in the library, they realized a savings of 250 man-days per month at a cost of 25 man-days in support and maintenance time. In Ada projects at NASA, 42 % of the projects' code was reused code and the productivity increase was 33.9 non-comment lines of code per staff day [Coome et al. 90]. With a library of 136 components consisting of 168,000 source code lines available to 700 programmers, GTE Data Services achieved a 14 % code reuse rate for new systems and a savings of 1.5 million dollars in 1987 [Todd 90]. [Banker and Kauffman 90] reported 65% reuse rate for 21 financial projects that were developed using a CASE tool. In this specialized environment, a productivity increase of 796 % in the second year was reported. However, it is difficult to determine how much of this productivity increase is due to reuse, and how much is due to the automated CASE environment and application generation facilities provided by the CASE tool.

Another benefit related to software reuse comes from higher quality and reduced need for testing. Quality is improved by reusing

software resources which are already tested and verified by use. The Fuchu Software Factory maintained an average program quality level of 2-3 faults per thousand lines of code [Matsumoto 87]. The Hartford Insurance Group reported that 17 software engineers and other staff people spent 20 hours to qualify each 1,000 lines of code accepted for reuse [Joyce 88]. One defect per thousand lines of code was reported in NASA's third Ada project [Coome et al. 90].

In summary, reuse rates in the range 14-65 % have been reported with productivity increases in the range 10-50 %. Despite this evidence of large increases in productivity, software reuse is not widespread for a number of technical, behavioral and economic reasons as discussed later in this paper.

4.2 Cost-benefit Models

In [Gaffney and Durek 89], several economic models of software reuse are presented. These model the impact of software reuse on development productivity relative to that obtained if the software product was to be built using all new code. Relationships among productivity, reuse rate, and cost are investigated in several different situations. Their theoretical model supports the order of magnitude of the results in Table 2. For example, when the reuse rate is 50% and the relative cost of reused software (compared to the cost of all new software) is 40 %, their formula shows a theoretical increase in productivity of 25%. Other economic models of software reuse to evaluate software development performance include [Banker et al. 90] and [Balda and Gustafson 90].

Papers on economic issues support the idea that software reuse can be a keystone in efforts to improve productivity and quality. However there are few papers about how to measure reuse rate and gains in productivity from software reuse when software development is considered as an ongoing activity. Cost for the installation of the methodology to support software reuse and maintenance cost are usually

not considered. Developing economic models of software reuse which consider software development as an ongoing activity in organizations is a needed future research direction. An economic model of software reuse that considers time in software development and costs for the methodology installation and maintenance, is needed to address measurement problems and provide analyses of software reuse approaches.

5. REUSE METHODOLOGIES

In the framework in Figure 1, technical issues are divided into two categories: reuse methodologies and software approaches. In this section, we group the challenges involved in developing a reuse methodology. Section 6 will discuss the various software approaches that can be used to support whatever methodology is adopted.

As the development costs of software systems increase, the role of reuse becomes more important in software engineering. For this reason, a software engineering methodology should support the notion of developing and leveraging reusable software resources [Rubin 90]. Developing a software methodology that supports reuse is an active focus of current research [Hall 89] [Freeman 87a] [Wirfs-Brock and Wilkerson 89].

Here, we look at reuse methodologies rather narrowly in terms of the process steps that might be performed by a software development group. Approaches to reusability that involve broader organizational strategies are discussed in section 7. Table 3 summarizes the reuse processes proposed by a number of researchers. Obviously, the researchers describe, or implicitly assume, essentially the same steps in the reuse process even though each emphasizes different processes. For example, [Prieto-Diaz and Freeman 87] emphasized the role of classification in code reuse. The decomposition/abstraction process proposed by Boldyreff is for the decomposition of large software systems into their component concepts and the abstraction of a reusable software concept that is generic from a number of similar specific

software concepts.

[Boldyreff 89]	[Redwine and Riddle 89]	[Biggerstaff and Richter 89]	[Prieto-Diaz and Freeman 87]
decomposition/ abstraction			
classification	cataloging	finding	accessing
selection	selection	finding	accessing
adaptation	adaptation	understanding, modifying	understanding, adapting
composition	assembly	composing	

Table 3. The Summary Table of the Proposed Reuse Processes

Because it cannot be expected that new software projects can be developed entirely by reusing existing reusable components, part of the target software systems will normally be developed from scratch. Therefore, traditional software development methodologies (such as the software development life cycle or prototyping) need to be expanded to support software development in part from scratch and in part from reuse as shown in Figure 2. Six processes are involved in developing a target software system. The first process involves classifying the existing software resources to be reused in the future. This has to be performed at the initiation of the reuse program to develop a library of software resources. It must also be performed each time a new reusable resource is to be cataloged. The remaining processes form part of the software development life cycle proper. The second process is for specifying requirements for the new system. This has to be performed regardless of whether the software component is to be developed from scratch or not. The third process is for identifying and selecting appropriate target software resources from reusable software resources based on the requirements specification. The fourth process, modifying software resources, is necessary when the library resources

retrieved do not exactly match the requirements specification. The fifth process, build new components, is necessary when there is no similar software resource in the existing reusable software resources for some of the requirements. Finally the sixth process is required to combine the new and reused software resources into the target software system.

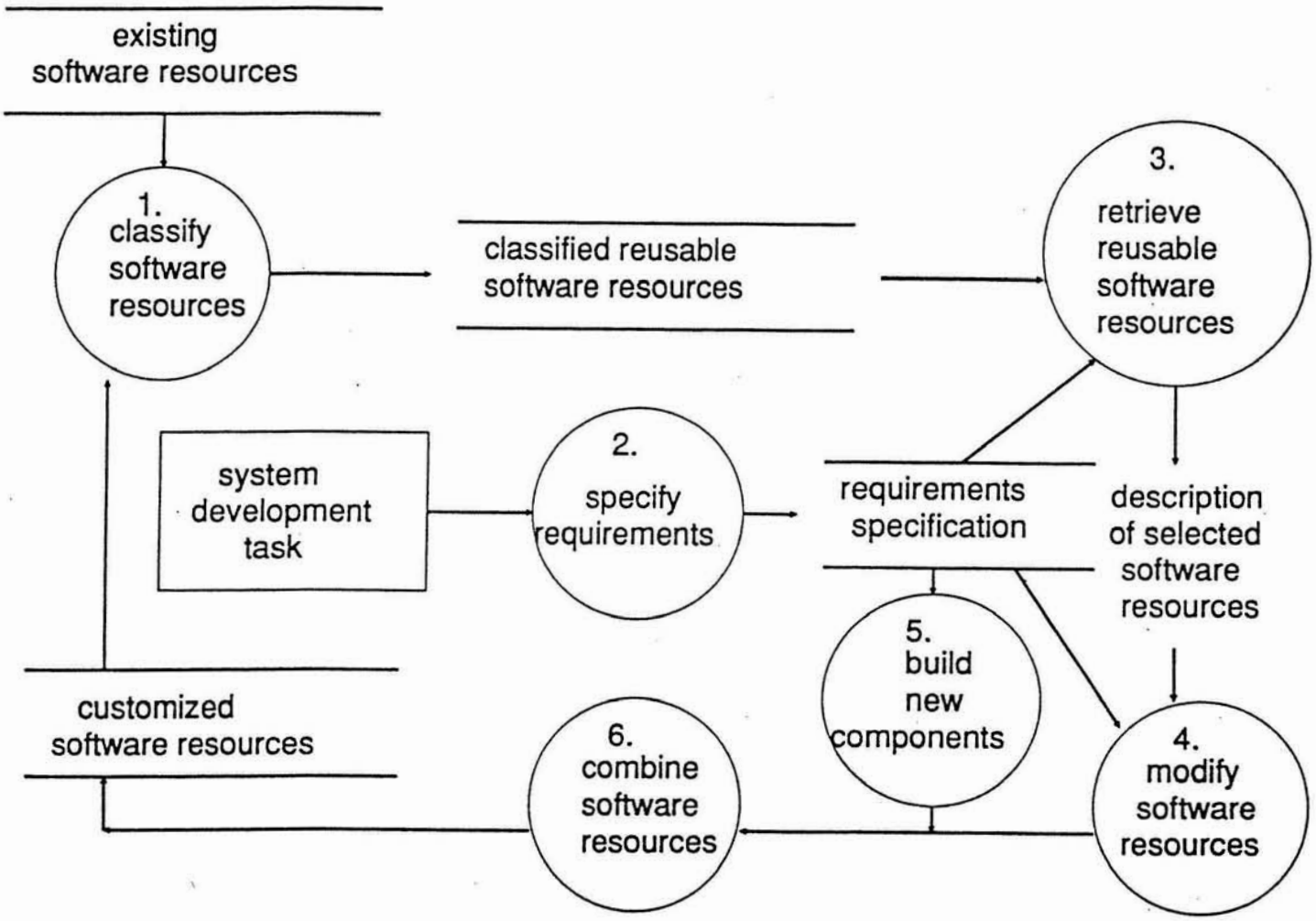

Figure 2. System Development with the Concept of Software Reuse

Note that the normal systems development life cycle involves only

Steps 2, 5, and 6 in Figure 2. For any project, the main gain from the reusability approach is given by the difference in the costs of performing Step 5 versus Steps 3 and 4. The cost of establishing and updating the reusability library (Step 1) has to be amortized over all projects.

Since we are dealing with reusable resources from all phases of the software development life cycle, the software resources that are retrieved in Figure 2 might be in the form of specifications, data flow diagrams, program structure charts, source code, or object code. If a strict life cycle approach is used, the steps in the above diagram might be iterated for each phase of the lifecycle in order to complete the specification for the entire target system (or a subsystem of the target system) that is relevant to that phase. Alternatively, the user might wish to retrieve the documentation for all lifecycle phases for a single reusable object at once. Both approaches seem to be useful a priori. Further research will be needed to investigate the applicability of these two strategies.

The steps in Figure 2 can be performed independently of the use of CASE tools or application generators. However, the use of CASE tools implies that information concerning all the phases of development is captured in the normal course of affairs. This implies that CASE tools provide a high potential for software reuse.

6. SOFTWARE APPROACHES

The many different software development approaches can be separated into four categories: generation methods, composition methods, object-oriented methods, and the CASE approach.

6.1 Generation Methods

The objects being reused are general problem solving patterns that drive the generation of the target programs. There are three classes of

generation methods: language-based systems, application generators, and transformation-based systems.

When a problem solution is generic and well understood, it may be possible to develop a language-based system that reuses previously developed software solutions. Language-based systems emphasize the notation that is to be used to describe the target system. The language types range from general purpose languages, called very high level languages (VHLL's) [Dubinsky et al. 89], to special purpose languages, called problem oriented languages (POL's) [Lee 86]. The major problem with language-based systems is that they are generally feasible only for domains with a very large number of users. Otherwise, the expense in developing the systems may not be justified. In addition, unless there is a broad domain of application, it may be difficult to get agreement on the language to be used and sufficient people to learn the language. Finally, the reuse problem remains at a different level - the objects developed in these languages, are themselves reusable.

Application generators are software packages which are designed to help end-users build applications in a given domain. Computer Aided Software Engineering(CASE) tools such as Texas Instrument's Information Engineering Facility(IEF), Knowledgeware's Information Engineering Workbench(IEW), and Seer Technologies' High Productivity Systems(HPS) contain code generators that produce executable code directly from the design specifications. Application generators differ from language-based systems in that code is generated from a higher level specification of the task in a nonprocedural language which is usually designed to be easier to use than procedural languages. In comparison with language-based systems, the domain of application for application generators has generally been more restricted. Application generators reuse built-in patterns of code to generate new systems. Again, the reusability problem is transformed to a different level - i.e., to the higher level objects in the design specification language

Transformation systems transform code written in one language to

code in another language. There are three areas where transformation systems are useful: (1) writing in a powerful language such as Lisp, then reusing the logic by transforming it to a language in which execution is more efficient (e.g. FORTRAN) [Boyle and Muralidharan 84] or more portable (e.g. C), (2) reusing software when hardware upgrades or operating system changes occur, and (3) migrating to a newer, more common language for reasons of standardization. A methodology and supporting programming environment that provides for reuse of abstract programs through refining a single abstract program to a family of distinct concrete programs are described by [Cheatham 84]. It is concluded that the reuse of abstract programs to do rapid prototyping and custom tailoring is a viable alternative to the conventional programming paradigm.

When several programs are to be derived from a single program, program transformation is economic. Transformation is also good for achieving portability in systems because porting the system to a new environment is simply handled. Transformation systems are in the early stage of development and not widely used at the present time.

The following table briefly summarizes the three generation methods.

Approaches	Characteristics
Language-based Systems	A special language is used to express common functions in a terse and elegant form
Application Generators	A special language is used to generate new software systems by modifying and reusing known patterns of software solutions
Transformation Systems	Reuse of the logic of existing software systems by transforming into a new language

Table 4. Summary of Three Approaches to Generation Methods

Advances in generation methods provide a method of attacking the problem of software productivity that may seem like an alternative to traditional reuse of existing resources. In fact, these approaches both codify reuse and are most effective when used in conjunction with composition methods of reuse as discussed below.

6.2 Composition Methods

Software development approaches that emphasize the composition approach utilize existing reusable resources that are viewed as atomic building blocks which are organized and combined according to well-defined rules. The major objective for these approaches is the creation of software libraries containing generic and reusable software components which can be combined to produce new target systems. This is the traditional view of reusability research. There are three areas of research emphasis: the development of application component libraries, the classification and retrieval strategies, and composition principles.

Application Component Libraries

In application component libraries, the components to be reused are largely atomic and are usually unchanged in the course of their reuse [Biggerstaff and Richter 89]. Examples of such components are subroutines, functions, programs, and Smalltalk-style objects [Lanergan and Grasso 89] [Cavaliere 89] [Goldberg and Robson 83]. Component libraries have been very effective for statistical and mathematical applications [SPSS 86] [IMSL 84], but they are not sufficient to achieve high improvement in other areas of software development. There are two main reasons. First, there is the difficulty of classifying reusable components in such a way that another person can easily identify and retrieve them. Second, once retrieved, the reusable components have to be combined into the target system. Software components in the library are written in a specific programming language, so detailed implementation decisions have already been made.

This means that there is little flexibility in reusing the software components. It is difficult to modify the basic structure of the components, or to use a part of the components.

Software Classification and Retrieval Strategies

Most approaches to the software classification and retrieval problem have their roots in traditional library systems. These approaches include full-text retrieval, keyword schemes, enumerative schemes, and the faceted approach.

In the full-text retrieval approach, target resources are retrieved by searching for all documents containing certain words which the user has specified [Harter 86]. STAIRS from IBM [Blair and Maron 85] is a typical example of full-text retrieval systems for traditional library applications. In the software reuse domain, the text that is searched can be either the source code itself or a short English description of the program or other reusable object as in CATALOG system from AT&T [Frakes and Nejmeh 87]. The full-text retrieval approach avoids the need for manual indexing which is costly and inconsistent. However, as the database becomes larger, it becomes difficult to search and retrieve all the relevant resources and nothing but the relevant resources.

Keyword approaches allow lists of keywords to be attached to each item as it is stored in the library. In many systems, these keywords come from a standard list. They may be listed by the author in any order. An example software library that uses the keyword approach is that used by the NASA/Ames Research Center [Jones and Prieto-Diaz 88].

Enumerative classification schemes take a subject area and divide it into categories hierarchically arranged with each item being assigned to one of the categories as it is stored in the library. A prominent example of this approach is the Dewey Decimal Scheme [Dewey 79] for library retrieval. Software libraries that use this

classification scheme include the International Mathematics and Scientific Library [IMSL 84]. An inherent problem with enumerative schemes is traversing the hierarchical tree to find the appropriate class [Jones and Prieto-Diaz 88].

The faceted method synthesizes new classes from a set of independent, elemental items called facets. Facets[1] are defined as the categories, perspectives, or viewpoints of a particular collection of concepts or a domain [Vickery 60]. For example, in [Jones and Prieto-Diaz 88], the software is described by five facets (Functional Area, Action, Object, Language, and Hardware). A standard list of terms for each facet is stored by the system. For example, terms for functional area include accounts-payable, accounts-receivable, billing, and budgeting. The process of classifying a document may create an entirely new class with a membership of one. Each item stored in the software resource library is described by the list of values it has in each facet. Advantages of the faceted method are that it allows multiple dimensions to be defined for relevant concepts and standardizes the vocabulary for these concepts. This method has been successfully used as a technique for software classification by a number of researchers on software reusability. Two different systems using essentially the same facets are described in [Ruble 87], and [Prieto-Diaz and Freeman 87], [Jones and Prieto-Diaz 88]. [Owen et al. 88] addressed the problem of providing tools for the storage and retrieval of reusable software components which consist of Ada packages and procedures. They focused on a faceted classification scheme method where each software module is described by a tuple of classes composed of descriptors from a controlled vocabulary. They developed prototypes which showed both the feasibility and the flexibility of the faceted classification scheme method.

---

[1]A different meaning of the term facet occurs in artificial intelligence. In artificial intelligence, facets mean the mechanism by which control information (such as permitted values for the slot and display format for the slot) can be attached to slots in a frame.

Other approaches to the classification/retrieval problem have been used in software reusability research. These are based on semantic representation and retrieval. [Mackellar and Maryanski 89] propose a retrieval mechanism which retrieves objects that are close to a user's description of the target data type from an existing library of data types by using matching and scoring rules. [Wood 88] concentrated on the effective storage and retrieval of reusable software components by developing a representation scheme based on Conceptual Dependency, a theory used to represent the meaning of natural language [Schank 75]. Software components are represented in terms of the relationships between functions and objects that occur in the context of functions.

The most comprehensive approach to the classification/retrieval problem is to build a specialized information system that records design and structural information about existing software systems. [Debanbu et al. 91] discuss the problem of complexity and "invisibility" as inhibitors of software reuse. By invisibility they mean that the structure of software is hidden and difficult to understand. They describe a system called LaSSIE (Large Software System Information Environment). LaSSIE uses frame-based knowledge representation and reasoning technology to promote reusability of a large software system. The LaSSIE system simplifies the knowledge engineer's task, while still providing semantic retrieval as well as a rich knowledge structure for browsing, navigation, and query reformulation.

In a test of retrieval effectiveness using the STAIRS (full-text retrieval) system in a legal application, the average recall ratio (i.e., the proportion of relevant retrieved items to the number of relevant items) was 20 % and the average precision ratio (i.e., the proportion of relevant retrieved items to the number of retrieved items) was 79 % [Blair and Maron 85]. This level of performance (especially the recall ratio) is disappointing. We are unaware of similar measures of retrieval effectiveness from experiments in software reuse. Factors in this domain that might improve performance

are as follows. First, the number of relevant objects for a given software retrieval request should be much lower than in a traditional library application - often there will only be one relevant resource if the request is at the concrete level of abstraction. Second, the domain of software reuse may be more structured with more potential for the use of standardized terms that would aid retrieval. On the other hand, there may be a higher probability that there are no relevant items in the reuse library that can satisfy a given request. This is because of the limited scope of software reuse libraries when compared with the diversity of possible software needs. Experiments on the effectiveness of the various classification-retrieval schemes mentioned above are needed to guide further development of software reuse methodologies.

Composition Principles

A number of researchers have emphasized the importance of software organization and composition principles by which components are combined into target programs. The UNIX pipe mechanism [Kernighan 84] provides a limited form of composition in which one program's outputs are connected to another program's inputs to construct more complex programs. Smalltalk uses message passing and inheritance as a composition principle [Goldberg and Robson 83]. [Katz et al. 87] developed the PARIS system which maintains a library of programs in which some parts remain abstract and undefined. They provide an interactive mechanism to search through the library for a schema that can be reused. Their approach provides another way to increase the flexibility of software reuse by replacing nonprogram abstract entities in the retrieved schemas with concrete programs.

There are several problems in existing approaches to organization and composition of software components. First, apart from the UNIX pipe mechanism (and the special case of application generators), the difficult problem of integrating reusable objects into the target system is usually left entirely to the users. Second, most existing software reuse approaches focus only on source code, not on various

resources from other stages of software development activity. Third, relationships among components are not explicitly represented; existing approaches consider only independent components. Fourth, they use detailed implementation criteria for classification and organization. Research and development of mechanisms that facilitate the combination of heterogeneous software resources is needed if the reuse potential of existing software resources is to be fully realized.

6.3 Object-Oriented Methods

Object-oriented programming languages provide another approach to reusability. A good discussion is contained in [CACM 90]. The properties of object oriented languages that help reusability include information hiding, property inheritance, and polymorphism. Information hiding is a reusability mechanism, since those parts of a system which cannot see information that must change can be reused to (re)build the system when that information does change. Property inheritance allows new subclasses to be built on top of superclasses by inheriting variables and methods of the superclass. The process of inheritance encourages reuse of previously defined data attributes and procedures in a more specific manner. Polymorphism means that operations have multiple meanings depending on the types of their arguments [Micallef 88]. Polymorphism can make reuse more flexible. [Tarumi et al. 88] have developed a programming environment for object-oriented programming which supports reuse of classes through the use of an expert system.

Object-oriented programming languages provide flexibility in using reusable objects. However, it is sometimes difficult to combine operations defined by different reusable objects. Even in an object-oriented environment, a major problem is that it is still difficult for users, especially those who were not involved in the development of the existing software resources, to know whether there are reusable software resources to match their needs. Moreover, organizations will continue to use traditional software development approaches for reasons of inertia and efficiency as well as because of the large installed

base of software that has to be maintained.

6.4 CASE Tools and Reuse

[Banker and Kauffman 90] report that the level of code reuse is the major factor that deserves attention in software projects developed using CASE (Computer-Aided Software Engineering) tools because extensive code reuse can increase productivity by an order of magnitude or more, and thus yield significant cost reductions in software development operations.

The central idea of CASE tools for reuse is the availability of a software base containing software and software-related constructs such as domain knowledge and methodological knowledge [Karakostas 89], [Czuchry and Harris 88]. The availability of a software base makes application-oriented software reuse from early phases of the software development cycle (such as analysis and design) feasible with CASE tools. In contrast, most other current reuse approaches support only independent single component reuse at the coding phase.

Two different aspects of the CASE approach, integrated data dictionaries and code generators, are reported to promote software reusability by [Oman 90]. The data dictionary integrates all reusable software resources from various tasks into the central data dictionary and facilitates access to these resources for reuse purposes. CASE tools such as Excelerator and Prosa provide an integrated data dictionary. Code generators associated with a number of CASE tools automatically generate target source code from graphical software system models. CASE tools such as Cradle, HPS, IEF, IEW, and Prosa have one or more code generators for programming languages such as Ada, C, Cobol, Pascal, and SQL.

While CASE tools facilitate software reuse, several aspects of these systems can be improved. These include the classification/retrieval method, the explicit representation of design

constraints, and assistance in the composition phase of reuse. If these capabilities can be added to CASE tools, software reuse can be extended to include reuse of customized application systems, and reuse of design objects from early phases of the software cycle.

7. NON-TECHNICAL ISSUES

Most research on software reusability deals with technical issues. Such research concentrates on the "WHAT and HOW" of the reuse issue, but rarely explains "WHY". Non-technical issues explain the difficulties that have been experienced in promoting widespread reuse in industry. They can be classified into two categories: psychological issues and organizational issues.

7.1 Psychological issues

By understanding the merits of existing software paradigms from the perspective of cognitive psychology, researchers [Tracz 79] [Curtis 83] try to provide insights into dealing with complex problem solving. A discussion of the psychological inhibitors identified with software reuse provides answers to the question "What makes reusing software artifacts difficult?" [Tracz 87]. Major inhibitors to software reuse include: the "Not Invented Here" syndrome, lack of trust in programming products, no clearly defined standards for developing reusable software, software complexity and the invisibility problem, few large repositories of reusable software, and few tools to help users understand and access the resources that might be available in those repositories.

[Soloway and Ehrlich 84] study empirically the differences between expert programmers and novices. They suggest that expert programmers have at least two types of knowledge: programming plans and rules of programming discourse. When experts develop applications, they try to match pieces of the problem with solution segments with which they are familiar [Soloway and Ehrlich 83]. This implies that portions

of designs are reused when applications are developed. Maintaining a database of such previous designs should facilitate performance and learning by software developers at all levels of proficiency.

The size limit of short term memory [Miller 56] is one of the chief factors in the problem of "invisibility", which is seen as an important deterrent to software reuse [Debanbu et al. 91]. The perceived complexity of software can be reduced by chunking or modularization of components. This argument is directly related to information hiding or abstraction and to the object-oriented approach [Parnas et al. 83].

Finally, conceptual differences that cause different people to describe the same thing in different ways [Bhargava and Beyer 91] are at the heart of the classification/retrieval problem in software reuse.

The above observations support the need for tools to reduce the complexity of expressing user requirements, and to assist users in finding reusable components and understanding the software systems they are attempting to build.

7.2 Organizational issues

Research based on an organizational view provides an understanding of why managers often do not adopt a reusable software engineering approach for software projects. First, there is the lack of a corporate infrastructure which encourages and rewards reuse [Barnes et al. 87]. Some organizations rely on incentive programs to stimulate programmer interest in reusability: the Hartford Insurance Group pays $300 for the best productivity suggestion of the month, while GTE pays authors a cash bonus of $25 each time a component is reused [Joyce 88]. Second, there is a lack of user training in reuse techniques [Horowitz and Munson 84]. Third, costs to create the tools and methodologies are generally not within the budget of a single project [Jones 86]. Fourth, higher degrees of reuse may lead to fewer experts. Any reduction in

headcount might be perceived as reducing the empire a manager commands [Rauch-Hindin 83].

[Karimi 90] proposed an asset-based systems development strategy that plans for software reuse at the organizational level through an integrated approach to systems development. He claimed that the traditional application-oriented approach to system development is inappropriate for developing reusable software parts. The asset-based systems development strategy requires that top management understands the critical role of software reuse, and project management and that software experts participate in strategic information systems planning. Because the asset-based method is based on data and process modeling, it can be integrated with the structured analysis and data modeling techniques of systems analysis and design. The method also supports both the functional and object-oriented systems development approaches.

Software reuse also implies security and legal problems. Increasing the reuse potential of software resources could facilitate access by competitors and decrease the competitive advantage of the original developers. Existing copy right and patent protections have proved to be of limited use in preventing unethical use of software. Research is needed on how to reconcile the conflicting objectives of easier access to reusable resources and protection from unauthorized use.

The papers on organizational issues stress that software reusability should be broadly defined to include any kind of reusability which achieves the desired benefits, i.e., reduced cost and time for software development and maintenance. They also point to the need for research on the social and organizational impacts of programs that promote reuse of software resources.

8. Conclusions and Research Directions

In the above, general, technical, and non-technical issues of

software reuse were broadly addressed. Software reuse is regarded as a key to improving software development productivity and quality [Tracz 87] [Gruman 88] [Biggerstaff and Richter 89]. As outlined above, researchers and practitioners have proposed many approaches to increase the potential of software reusability. The definition of widespread software reuse developed in this paper places equal emphasis on data and process resources, abstract and concrete resources, specific application-oriented and generic function-oriented resources. It also emphasizes a wide range of tasks from maintenance of existing systems to development of new systems.

The field of software reuse is at a formative stage. Major research opportunities exist in all of the areas of software reusability research that are depicted in Figure 1. These research questions have been suggested in the body of the paper. Briefly, we need better tools for modeling the productivity gains from reuse to help motivate its adoption and guide research into fruitful directions. We need to understand better how to build reusability into our software development methodologies. In the technical area, we need to understand the role of reusability in the generation, composition, object-oriented, and CASE approaches to software development. A key will be to develop improved methods for classifying, organizing, and retrieving software resources. Finally, to make the technical solutions work, we need to know how to build a supportive organizational environment and to solve the psychological problems of motivation and bounded rationality.

In our opinion, the key conclusion that can be drawn from this review is that reuse needs to be viewed in the context of a total systems approach. Thus, we do not envisage only libraries of useful data definitions, software routines, or objects that can be reused by a motivated user. Rather, we also envision a software system or reuse support system(RSS) that helps document and elucidate existing application systems so that the ideas and design decisions involved in their creation can be reused either in the context of maintenance or

when building new systems. In the latter case, the reuse support system should encourage the use of standard data definitions, and software design approaches both through the organization and also between organizations. The LaSSIE system [Debanbu et al. 91] and Telos [Mylopoulos et al. 90] are two approaches to building such an "information system about an information system".

References

[Banker et al. 90]
Banker, R.D., Kauffman, R.J., and Zweig, D.
Metrics for the Code Reuse in Software Development.
Working Paper, 1990.

[Banker and Kauffman 90]
Banker, R.D. and Kauffman, R.J.
An Empirical Assessment of CASE Technonology: A Study of Productivity, Reuse, and Functionality.
Working Paper, 1990.

[Balda and Gustafson 90]
Balda, D. and Gustafson, D.
Cost Estimation Models for the Reuse and Prototype Software Development Life-Cycles.
ACM SIGSOFT Software Engineering Notes Vol. 15, No. 3, Pages 42-50, July, 1990.

[Barnes et al. 87]
Barnes, B. et al.
A Framework and Economic Foundation for Software Reuse,
Tutorial: Software Reuse: Emerging Technology,
IEEE Computer Society, EHO278-2, pages 77-88.

[Bhargava and Beyer 91]
Bhargava, H. and Beyer, R.
Automated Detection of Naming Conflicts in Schema Integration: Experiments and Quiddities.
Technical Report, Naval Postgraduate School, NPS-AS-91-011.

[Biggerstaff and Richter 89]
Biggerstaff, T.J., and Richter, C.
Reusability Framework, Assessment, and Directions.
Software Reusability Vol I Concepts and Models.
Addison Wesley, 1989.

[Blair and Maron 85]
Blair, D. and Maron, M.
An Evaluation of Retrieval Effectiveness for a Full-text Document-retrieval System.
Comm. of the ACM Vol. 28, No. 3, pages 289-299, March 1985.

[Boldyreff 89]
Boldyreff, C.
Reuse, Software Concepts, Descriptive Methods, and the Practitioner Project.
ACM SIGSOFT Software Engineering Notes Vol. 14, No. 2, pages 25-31, April, 1989.

[Boyle and Muralidharan 84]
Boyle, J.M., and Muralidharan, M.N.
Program Reusability through Program Transformation.
IEEE Transactions on Software Engineering SE-10(5):574-588,
September, 1984.

[CACM 90]
Communications of the ACM.
Series of Articles on Object-oriented Programming,
Sept., 1990.

[Cavaliere 89]
Cavaliere, M. J.
Reusable Code at the Hartford Insurance Group.
In Software Reusability. Vol. 2. Applicatins and
Experience.
Addison-Wesley, 1989.

[Cheatham 84]
Cheatham, T.E.
Reusability Through Program Transformation.
IEEE Transactions on Software Engineering SE-10(5):589-594,
September, 1984.

[Conte 88]
Conte, P.
Recycling Your Software.
Computer Language Vol. 5, No. 6, page 43,
June, 1988.

[Coome et al. 90]
Coome, T.N., Comer, J.R., and Rodjak, D.J.
Developing Reusable Software for Military Systems - Why It
is Needed and Why It isn't Working.
ACM SIGSOFT Software Engineering Notes Vol 15, No. 3,
pages 33-38, July, 1990.

[Curtis 83]
Curtis, B.
Cognitive Issues in Reusability.
In Proceedings of ITT Workshop on Reusability in
Programming, 1983.

[Czuchry and Harris 88]
Czuchry, A. and Harris, D.
KBRA:A New Paradigm for Requirement Engineering.
IEEE EXPERT, Winter 1988.

[Devanbu et al. 91]
Devanbu, P., Brachman, R., Selfridge, P, and Bsllsrd, B.

LaSSIE: a Knowledge-based Software Information System.
Comm. of ACM, May 1991.

[Dewey 79]
Dewey, M.
Decimal Classification and Retrieval Index.
Forest Press Inc., 19th Edition, 1979.

[DOD 86]
Office of Secretary of Defence.
Software Technology for Adaptable Reliable Systems - STARS.
Technical Program Plan.
Washington D.C., August 1986.

[Dubinsky et al. 89]
Dubinsky, E., Freudenberger, S., Schonberg, E., and
Schwartz, J.T.
Reusability of Design for Large Software Systems:An
Experiment with the SETL Optimizer.
Software Reusability Vol I Concepts and Models.
Addison Wesley, 1989. pages 275-293.

[Frakes and Nejmeh 87]
Frakes, W. and Nejmeh, B.
Software Reuse Through Information Retrieval
Proceedings of the Twentieth Annual Hawaii International
Conference on System Science, pages 530-535. Jan. 1987.

[Frank 81]
Frank, W.L.
Software Productivity:Any Breakthrough?
Computer World, July, 1981.

[Freeman 87]
Freeman, P.
Tutorial: Software Reusability
IEEE Computer Society Press, 1987.

[Freeman 87a]
Freeman, P.
A Conceptual Analysis of the Draco Approach to Constructing
Software Systems,
IEEE Tutorial: Software Reusability,
IEEE Computer Society Press, 1987.

[Fylstra and Gill 80]
Fylstra, D. and Gill, M.
VISICALC, Personal Software, Inc.
Part V, pages 1-13, 1980.

[Gaffney and Durek 89]
Gaffney, J.E. Jr, and Durek, T.A.

Software reuse - key to enhanced productivity:some quantitative models.
Information Software Technology Vol. 31, No. 5, pages 258-267, June, 1989.

[Goldberg and Robson 83]
Goldberg, A., and Robson, D.
Smalltalk 80:The language and Its Implementation.
Addison-Wesley, 1983.

[Gruman 88]
Gruman, G.
Early Reuse Practice Lives up to Its Promise.
IEEE Software pages 87-91, November, 1988.

[Hall 89]
Hall, P.
A Metamodel for Software Components and Reuse,
Practitioner Project Working Paper,
P1094-BrU-PH-WPBI-WORKING PAPER-0027, January 1989.

[Harter 86]
Harter, S.
Online Information Retrieval Concepts, Principles, and Techniques
Academic Press, Inc., 1986.

[Horowitz and Munson 84]
Horowitz, E. and Munson, J.B.
An Expansive View of Reusable Software.
IEEE Transaction on Software Engineering SE-10(5):477-487, September, 1984.

[IMSL 84]
International Mathematics and Scientific Library
10th Edition, IMSL Inc., 1984.

[Jones 84]
Jones, T.C.
Reusability in Programming:A Survey of the State of the Art.
IEEE Transaction on Software Engineering SE-10(5):488-493, September, 1984.

[Jones 86]
Jones, T.C.
Programming Productivity.
McGraw-Hill, 1986.

[Jones and Prieto-Diaz 88]
Jones, G. and Prieto-Diaz, R.
Building and Managing Software Libraries.
In Proceedings of IEEE COMPSAC, pages 228-236, 1988.

[Joyce 88]
Joyce, E.J.
Reusable Software:Passage to Productivity?
Datamation pages 97-102, Sep., 15, 1988.

[Karakostas 89]
Karakostas, V.
Requirements for CASE Tools in Early Software Reuse.
ACM SIGSOFT Software Engineering Notes Vol. 14 No. 2,
Pages 39-41, April, 1989.

[Karimi 90]
Karimi, J.
An Asset-based Systems Development Approach to Software Reusability.
MIS Quarterly:179-198, June, 1990.

[Katz and et al. 87]
Katz, S., Richter, C.H., and The, K.
PARIS:A System for Reusing Partially Interpreted Schemas.
In Proceedings of IEEE 9th International Conference on Software Engineering, pages 377-385, 1987.

[Kernighan 84]
Kernighan, B.W.
The Unix System and Software Reusability.
IEEE Transaction on Software Engineering SE-10(5):513-518,
September, 1984.

[Lanergan and Grasso 89]
Lanergan, R.G. and Grasso, C. A.
Software Engineering with Reusable Designs and Code
In Software Reusability. Vol. 2, Applications and Experience.
Addison-Wesley, 1989.

[Lee 86]
Lee, J.S.
ALESA:A Language for Equation Structure Abstraction.
Technical Report, Dept. of Decision Sciences,
The Wharton School, Univ. of PA, 1986.

[Love 88]
Love, Tom.
The Economics of Reuse.
In Proceedings of IEEE COMPCON, pages 238-241, 1988.

[Lubars 86]
Lubars, M.
Affording Higher Reliability Through Software Reusability
ACM SIGSOFT Software Engineering Notes, Vol 11, No 5,
Oct. 1986

[Mackellar and Maryanski 89]
Mackellar, B.K. and Maryanski, F.
A Knowledge Base for Code Reuse by Similarity.
In Proceedings of IEEE COMPSAC, pages 634-641, 1989.

[Matsumoto 87]
Matsumoto, Y.
A Software Factory:An Overall Approach to Software Production.
Tutorial: Software Reusability
IEEE Computer Society Press, 1987.

[Micallef 88]
Micallef, J.
Encapsulation, Reusability and Extensibility in Object-Oriented Programming.
Journal of Object Oriented Programming:12-34, April/May, 1988.

[Miller 56]
Miller, G.A.
The Magical Number Seven Plus or Minus Two:Some Limits on Our Capacity to Process Information.
Psychological Review 63:81-97, 1956.

[Myers 87]
Myers, W.
ADA: First Users Pleased: Perspective Users Still Hesitate,
IEEE Computer Magazine, Vol 20, No. 3, March 1987.

[Mylopoulos et al. 90]
Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M.
Telos:Representing Knowledge About Information Systems.
ACM Tr. on Information Systems, Vol 8, No. 4, October 1990, Pages 325-362

[Oman 90]
Oman, P.W.
CASE Analysis and Design Tools.
IEEE Software: 37-44, May 1990.

[Owen et al. 88]
Owen, G.S., Gagliano, R., and Honkanen, P.
Tools for the Storage and Retrieval of Reusable MIS Software in ADA.
In Proceedings of ACM 16th Computer Science Conference, pages 535-539, 1988.

[Parnas et al. 83]
Parnas, D.L., Clements, P.C., and Weiss, D.M.
Enhance Reusability with Information Hiding.
In Proceedings of ITT Workshop on Reusability in

Programming, 1983.

[Prieto-Diaz and Freeman 87]
Prieto-Diaz, R. and Freeman, P.
Classifying Software for Reusability.
IEEE Software, Vol. 4, No.1, pp. 6-16, January, 1987.

[Rauch-Hindin 83]
Rauch-Hindin, W.B.
Reusable Software.
Electronic Design 31(3):176-193, Feb., 1983.

[Redwine and Riddle 89]
Redwine, S. T. jr. and Riddle, W. E.
Software Reuse Processes
In Proceedings of ACM Software Process Workshop,
pages 133-135, 1989

[Rubin 90]
Rubin, K.
Reuse in Software Engineering:An Object-Oriented
Perspective.
In Proceedings of IEEE COMPCON, pages 340-346, 1990.

[Ruble 87]
Ruble, D. L.
A Classification Methodology and Retrieval Model to Support
Software Reuse.
PhD Dissertation, 1987.

[Schank 75]
Schank, R. C.
Conceptual Information Processing.
North-Holland, Amsterdam, 1975.

[Seppanen 87]
Seppanen, V.
Reusability in Software Engineering.
Tutorial: Software Reusability
IEEE Computer Society Press, 1987.

[Soloway and Ehrlich 83]
Soloway, E. and Ehrlich, K.
What Do Programmers Reuse? Theory and Experiment.
In Proceedings of ITT Workshop on Reusability in
Programming, 1983.

[Soloway and Ehrlich 84]
Soloway, E. and Ehrlich, K.
Empirical Studies of Programming Knowledge.
IEEE Transactions on Software Engineering SE 10(5):595-609,
September, 1984.

[SPSS 86]
SPSSX User's Guide
McGraw Hill, 1986.

[Standish 84]
Standish, A.
An Essay on Software Reuse.
IEEE Transactions on Software Engineering SE 10(5):494-497, September, 1984.

[Sundfor 83]
Sundfor, S.
Reusable Software Engineering: A Survey of Concepts and Approaches.
Univ. of California, Irvine, Dept. of Information and Computer Science, RTP017, 1983.

[Tarumi et al. 88]
Tarumi, H.m Agusa, K., and Ohno, Y.
A Programming Environment Supporting Reuse of Object-Oriented Software.
In proceedings of IEEE 10th International Conference on Software engineering, pages 265-273, 1988.

[Todd 90]
Todd, D.
Code recycling:Reuse of Software can Save on Development.
Information Week, pages 50-51, May 14, 1990.

[Tracz 79]
Tracz, W. J.
Computer Programming and Human Thought Process.
Software-Practice and Experience 9():127-137, 1979.

[Tracz 87]
Tracz, W. J.
Software Reuse:Motivators and Inhibitors.
In Proceedings of IEEE COMPCON, pages 358-363, 1987.

[Tracz 90]
Tracz, W. J.
Where Does Reuse Start?
ACM SIGSOFT Software Engineering Notes,
Vol. 15, No. 2, pages 42-46, April 1990.

[Vickery 60]
Vickery, B.
Faceted Classification:A Guide to Construction and Use of Special Schemes.
Aslib, London, 1960.

[Wegner 90]
Wegner, P.
Concepts and Paradigms of Object-Oriented Programming
OOPS Messenger, Vol. 1, No. 1, August 1990.

[Wirfs-Brock and Wilkerson 89]
Wirfs-Brock, R. and Wilkerson, B.
Object-Oriented Design: A Responsibility-Driven Approach.
OOPSLA '89 Conference Proceedings,
Special Issue of SIGPLAN Notices, Vol. 24, No. 10, Oct. 1989

[Wood 88]
Wood, M.I.
Component Descriptor Frames: A Representation to Support
the Storage and Retrieval of Reusable Software Components.
PhD Dissertation, 1988.